BE CSE – IV Semester

18CSES402-Design and Analysis of Algorithms

UNIT 1

What is an algorithm?

Algorithm is a set of steps to complete a task.

For example,

Task: to make a cup of tea.

Algorithm:

1. add water and milk to the kettle,
2. boilit, add tea leaves,
3. Add sugar, and then serve it in cup.

What is Computer algorithm?

‘’a set of steps to accomplish or complete a task that is described precisely enough that a computer can run it’’.

An algorithm should have the following characteristics −

- Unambiguous − Algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning.
- Input − An algorithm should have 0 or more well-defined inputs.
- Output − An algorithm should have 1 or more well-defined outputs, and should match the desired output.
- Finiteness − Algorithms must terminate after a finite number of steps.
- Feasibility − Should be feasible with the available resources.
- Independent − An algorithm should have step-by-step directions, which should be independent of any programming code.

Algorithm Complexity

Suppose X is an algorithm and n is the size of input data, the time and space used by the algorithm X are the two main factors, which decide the efficiency of X.

- Time Factor − Time is measured by counting the number of key operations such as comparisons in the sorting algorithm.
- Space Factor − Space is measured by counting the maximum memory space required by the algorithm.

The complexity of an algorithm f(n) gives the running time and/or the storage space required by the algorithm in terms of n as the size of input data.

Space Complexity

Space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle. The space required by an algorithm is equal to the sum of the following two components −

1. A fixed part that is a space required to store certain data and variables,that are independent of the size of the problem. For example, simple variables and constants used, program size, etc.
2. A variable part is a space required by variables, whose size depends on the size of the problem. For example, dynamic memory allocation, recursion stack space, etc.

Space complexity S(P) of any algorithm P is S(P) = C + S(I), where C is the fixed part and S(I) is the variable part of the algorithm, which depends on instance characteristic I. Following is a simple example that tries to explain the concept −

Algorithm: SUM(A, B)

```
Step 1 -  START
Step 2 -  C ← A + B + 10
Step 3 -  Stop
```

Here we have three variables A, B, and C and one constant. Hence S(P) = 1 + 3. Now, space depends on data types of given variables and constant types and it will be multiplied accordingly.

Time Complexity

Time complexity of an algorithm represents the amount of time required by the algorithm to run to completion. Time requirements can be defined as a numerical function T(n), where T(n) can be measured as the number of steps, provided each step consumes constant time.

For example, addition of two n-bit integers takes n steps. Consequently, the total computational time is T(n) = c ∗ n, where c is the time taken for the addition of two bits. Here, we observe that T(n) grows linearly as the input size increases.

Asymptotic analysis of an algorithm

Asymptotic analysis of an algorithm refers to defining the mathematical boundation/framing of its run-time performance. Using asymptotic analysis, we can very well conclude the best case, average case, and worst case scenario of an algorithm.

The time required by an algorithm falls under three types −

Best Case − Minimum time required for program execution.

Average Case − Average time required for program execution.

Worst Case − Maximum time required for program execution.

Asymptotic Notations

Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

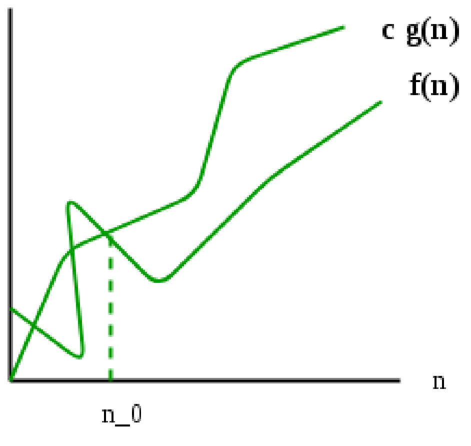
- **O Notation**
- **Ω Notation**
- **θ Notation**

Big Oh Notation, O

The notation O(n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.

The Big O notation is useful when we only have upper bound on time complexity of an algorithm. Many times we easily find an upper bound by simply looking at the algorithm.

```
O(g(n)) = { f(n): there exist positive constants c and
            n0 such that 0 <= f(n) <= c*g(n) for all n >= n0}
```



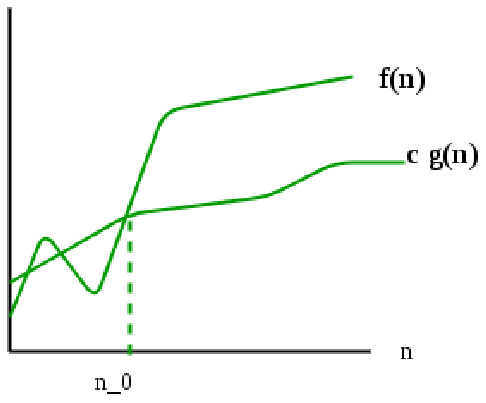
f(n) = O(g(n))

Omega Notation, Ω

The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.

For a given function g(n), we denote by Ω(g(n)) the set of functions.

Ω (g(n)) = {f(n): there exist positive constants c and
n0 such that 0 <= c*g(n) <= f(n) for all n >= n0}.



f(n) = Omega(g(n))

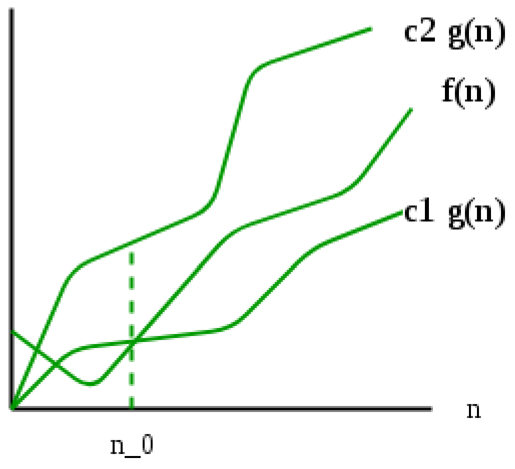
Theta Notation, θ

The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time. It is represented as follows −

For a given function g(n), we denote Θ(g(n)) is following set of functions.

Θ(g(n)) = {f(n): there exist positive constants c1, c2 and n0 such
that 0 <= c1*g(n) <= f(n) <= c2*g(n) for all n >= n0}

The above definition means, if f(n) is theta of g(n), then the value f(n) is always between c1*g(n) and c2*g(n) for large values of n (n >= n0). The definition of theta also requires that f(n) must be non-negative for values of n greater than n0.



f(n) = theta(g(n))

Recursive Algorithms and Recurrence Equations

- Performance of recursive algorithms typically specified with recurrence equations
- Recurrence Equations are also called as Recurrence and Recurrence Relations
- Recurrence relations have specifically to do with sequences (eg Fibonacci Numbers)

Analyzing Performance of Non-Recursive Routines is (relatively) Easy

Loop: $T(n) = \Theta(n)$

```
for i in 1 .. n loop
```

Loop: $T(n) = \Theta(n^2)$

```
for  i in 1 .. n loop

    for j in 1 .. n loop

    end loop;

end loop;
```

Analyzing Recursive Routines

- Analysis of recursive routines is not as easy: consider factorial

```
fac(n) is
    if $n$ = 1 then return 1
    else return fac($n$-1) * 1
```

Recurrences

- A recurrence defines $T(n)$ in terms of T for smaller values
- Example: $T(n) = T(n-1) + 1$
    - $T(n)$ is defined in terms of $T(n-1)$
- Recurrences are used in analyzing recursive algorithms
- Also known as: Recurrence Equation, Recurrence Relation

Substituting Up and Down

- Problem: Find value of T(n) = T(n-1) + 1 for n=4, with initial condition T(1)=2
- Substituting up from T(1):
    - T(1) = 2, Initial condition
    - T(2) = T(1) + 1 = 2+1 = 3
    - T(3) = T(2) + 1 = 3+1 = 4
    - T(4) = T(3) + 1 = 4+1 = 5
- Subsituting down from T(4):
    - Example: T(4) = T(3) + 1 = [T(2) + 1] + 1 = [[T(1) + 1] + 1] + 1 = 2+1+1+1 = 5

Fibonacci

- Algorithm:

```
fib(n)
    if n in 1 .. 2 return 1
    else return fib(n-1) + fib(n-2)
```

$$\boldsymbol{T(n)=T(n-1)+T(n-2)+1}$$

More Example Algorithms and their Recurrence Equations

- Factorial (Every Case): $T(n)=T(n-1)+1$
- Fibonacci (Every Case): $T(n)=T(n-1)+T(n-2)+1$
- Binary Search (Worst Case): $T(n)=T(n/2)+1$
- Quick Sort (Worst Case): $T(n)=T(n-1)+\Theta(n)$

- Quick Sort(Best Case): $T(n)=2T(n2)+\Theta(n)$
- Merge Sort(Every Case): $T(n)=2T(n2)+\Theta(n)$

Recursion Tree - *Follow the URLs*

https://lec.inf.ethz.ch/mavt/informatik/2018/dl/exercises/week09/slides09_recursion_trees.pdf

https://www.youtube.com/watch?v=4V30R3I1vLI

https://www.youtube.com/watch?v=IawM82BQ4II

https://www.youtube.com/watch?v=MhT7XmxhaCE

UNIT 4 AND UNIT 5

Difference between tractable and intractable problems.

Problems that can be solved in polynomial time are called tractable and the problems that cannot be solved in Polynomial time are called intractable.

Examples of tractable problems Searching an unordered list

- Searching an ordered list
- Sorting a list
- Multiplication of integers (even though there's a gap)
- Finding a minimum spanning tree in a graph (even though there's a gap)

Examples of some Intractable Problems

- Traveling Salesman Problem
- Knapsack Problem
- Bin Packing
- Job Shop Scheduling

Problem Reduction: Definition

- To solve an instance of problem A:
    - Transform the instance of problem A into an instance of problem B
    - Solve the instance of problem B
    - Transform the solution to problem B into a solution of problem A

We say that problem A **reduces to** problem B

P and NP problems.

P versus NP (polynomial versus nondeterministic polynomial) refers to a theoretical question presented in 1971 by Leonid Levin and Stephen Cook, concerning mathematical problems that are easy to solve (P type) as opposed to problems that are difficult to solve (NP type).

Any P type problem can be solved in "polynomial time." (A polynomial is a mathematical expression consisting of a sum of terms, each term including a variable or variables raised to a power and multiplied by a coefficient.) A P type problem is a polynomial in the number of bits that it takes to describe the instance of the problem at hand. An example of a P type problem is finding the way from point A to point B on a map. An NP type problem requires vastly more time to solve than it takes to describe the problem. An example of an NP type problem is breaking a 128-bit digital cipher. The P versus NP question is important in communications, because it may ultimately determine the effectiveness (or ineffectiveness) of digital encryption methods.

An NP problem defies any brute-force approach at solution, because finding the correct solution would take trillions of years or longer even if all the supercomputers in the world were put to the task. Some mathematicians believe that this obstacle can be surmounted by building a computer capable of trying every possible solution to a problem simultaneously. This hypothesis is called P equals NP. Others believe that such a computer cannot be developed (P is not equal to NP). If it turns out that P equals NP, then it will become possible to crack the key to any digital cipher regardless of its complexity, thus rendering all digital encryption methods worthless.

P, NP, NP complete and NP hard with example :

P -P is a complexity class that represents the set of all decision problems that can be solved in polynomial time. That is, given an instance of the problem, the answer yes or no can be decided in polynomial time.

Example

Given a connected graph G, can its vertices be coloured using two colours so that no edge is monochromatic?

Algorithm: start with an arbitrary vertex, color it red and all of its neighbours blue and continue. Stop when you run out of vertices or you are forced to make an edge have both of its endpoints be the same color.

NP -NP is a complexity class that represents the set of all decision problems for which the instances where the answer is "yes" have proofs that can be verified in polynomial time.

This means that if someone gives us an instance of the problem and a certificate (sometimes called a witness) to the answer being yes, we can check that it is correct in polynomial time.

Example

Integer factorisation is in NP. This is the problem that given integers n and m, is there an integer f with $1 < f < m$, such that f divides n (f is a small factor of n)?

This is a decision problem because the answers are yes or no. If someone hands us an instance of the problem (so they hand us integers n and m) and an integer f with $1 < f < m$, and claim that f is a factor of n (the certificate), we can check the answer in polynomial time by performing the division n / f.

NP-Complete -NP-Complete is a complexity class which represents the set of all problems X in NP for which it is possible to reduce any other NP problem Y to X in polynomial time.

Intuitively this means that we can solve Y quickly if we know how to solve X quickly. Precisely, Yis reducible to X, if there is a polynomial time algorithm f to transform instances y of Y to instances $x = f(y)$ of X in polynomial time, with the property that the answer to y is yes, if and only if the answer to f(y) is yes.

Example

3-SAT (Boolean satisfiability problem) - This is the problem wherein we are given a conjunction (ANDs) of 3-clause disjunctions (ORs), statements of the form

1. (x_v11 OR x_v21 OR x_v31) AND
2. (x_v12 OR x_v22 OR x_v32) AND
3. ... AND
4. (x_v1n OR x_v2n OR x_v3n)

where each x_vij is a boolean variable or the negation of a variable from a finite predefined list (x_1, x_2, ... x_n).

It can be shown that every NP problem can be reduced to 3-SAT. The proof of this is technical and requires use of the technical definition of NP (based on non-deterministic Turing machines). This is known as Cook's theorem.

What makes NP-complete problems important is that if a deterministic polynomial time algorithm can be found to solve one of them, every NP problem is solvable in polynomial time (one problem to rule them all).

NP-hard -Intuitively, these are the problems that are at least as hard as the NP-complete problems. Note that NP-hard problems do not have to be in NP, and they do not have to be decision problems.

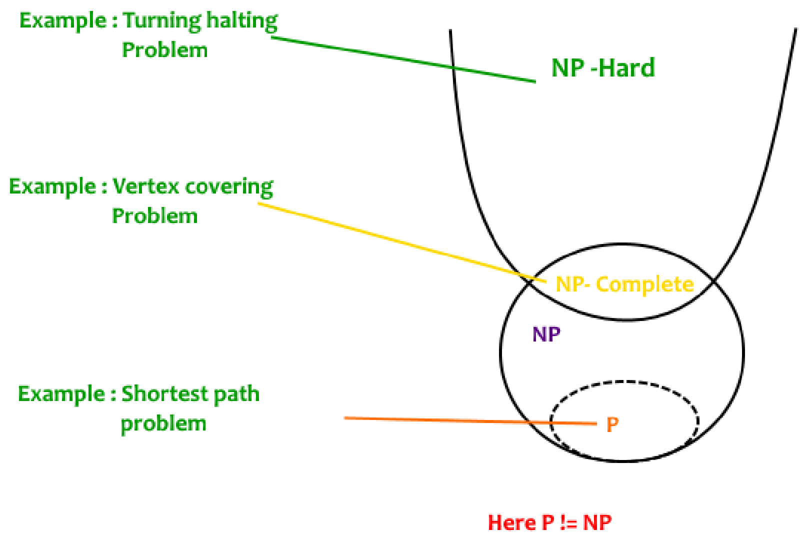
The precise definition here is that a problem X is NP-hard, if there is an NP-complete problem Y, such that Y is reducible to X in polynomial time.

But since any NP-complete problem can be reduced to any other NP-complete problem in polynomial time, all NP-complete problems can be reduced to any NP-hard problem in polynomial time. Then, if there is a solution to one NP-hard problem in polynomial time, there is a solution to all NP problems in polynomial time.

Example

The halting problem is an NP-hard problem. This is the problem that given a program P and input I, will it halt? This is a decision problem but it is not in NP. It is clear that any NP-complete problem can be reduced to this one. As another example, any NP-complete problem is NP-hard.

My favorite NP-complete problem is the Minesweeper problem


Example : Turning halting Problem
NP -Hard
Example : Vertex covering Problem
NP- Complete
NP
Example : Shortest path problem
P
Here P != NP

P versus NP

Every decision problem that is solvable by a deterministic polynomial time algorithm is also solvable by a polynomial time non-deterministic algorithm.

All problems in P can be solved with polynomial time algorithms, whereas all problems in NP - P are intractable. It is not known whether **P = NP**. However, many problems are known in NP with the property that if they belong to P, then it can be proved that P = NP.

If **P ≠ NP**, there are problems in NP that are neither in P nor in NP-Complete.

The problem belongs to class **P** if it's easy to find a solution for the problem. The problem belongs to **NP**, if it's easy to check a solution that may have been very tedious to find.

Approximation Algorithm

An Approximate Algorithm is a way of approach NP-Completeness for the optimization problem. This technique does not guarantee the best solution. The goal of an approximation algorithm is to come as close as possible to the optimum value in a reasonable amount of time which is at the most polynomial time. Such algorithms are called approximation algorithm or heuristic algorithm.

- For the traveling salesperson problem, the optimization problem is to find the shortest cycle, and the approximation problem is to find a short cycle.
- For the vertex cover problem, the optimization problem is to find the vertex cover with fewest vertices, and the approximation problem is to find the vertex cover with few vertices.

Randomized Algorithms

An algorithm that uses random numbers to decide what to do next anywhere in its logic is called Randomized Algorithm. For example, in Randomized Quick Sort, we use random number to pick the next pivot (or we randomly shuffle the array). Typically, this randomness is used to reduce time complexity or space complexity in other standard algorithms.

Randomized algorithms are classified in two categories.

Las Vegas: These algorithms always produce correct or optimum result. Time complexity of these algorithms is based on a random value and time complexity is evaluated as expected value. For example, Randomized QuickSort always sorts an input array and expected worst case time complexity of QuickSort is O(nLogn).

Monte Carlo: Produce correct or optimum result with some probability. These algorithms have deterministic running time and it is generally easier to find out worst case time complexity. For example this implementation of Karger's Algorithm produces minimum cut with probability greater than or equal to 1/n2 (n is number of vertices) and has worst case time complexity as O(E). Another example is Fermet Method for Primality Testing.

Example to Understand Classification:

Consider a binary array where exactly half elements are 0 and half are 1. The task is to find index of any 1.

A Las Vegas algorithm for this task is to keep picking a random element until we find a 1. A Monte Carlo algorithm for the same is to keep picking a random element until we either find 1 or we have tried maximum allowed times say k.

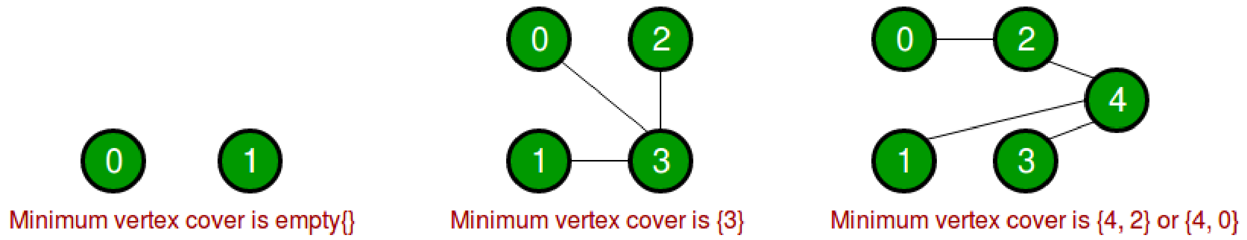
The Las Vegas algorithm always finds an index of 1, but time complexity is determined as expect value. The expected number of trials before success is 2, therefore expected time complexity is O(1).

The Monte Carlo Algorithm finds a 1 with probability $[1 – (1/2)^k]$. Time complexity of Monte Carlo is O(k) which is deterministic.

Vertex Cover Problem

A vertex cover of an undirected graph is a subset of its vertices such that for every edge (u, v) of the graph, either 'u' or 'v' is in vertex cover. Although the name is Vertex Cover, the set covers all edges of the given graph. Given an undirected graph, the vertex cover problem is to find minimum size vertex cover.

Following are some examples.

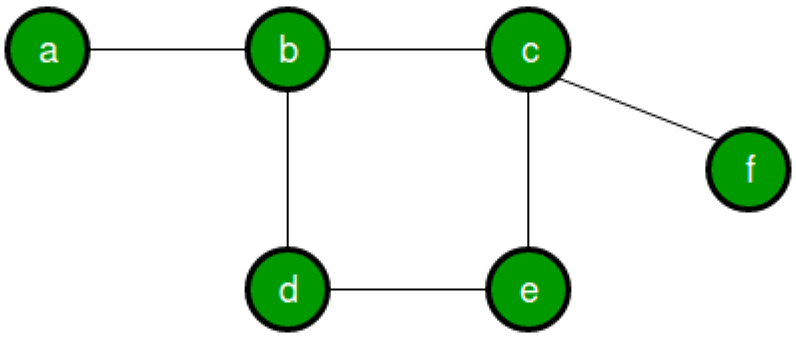

0
1
Minimum vertex cover is empty{}
0
2
1
3
Minimum vertex cover is {3}
0
2
4
1
3
Minimum vertex cover is {4, 2} or {4, 0}

Vertex Cover Problem is a known NP Complete problem, i.e., there is no polynomial time solution for this unless P = NP. There are approximate polynomial time algorithms to solve the problem though.

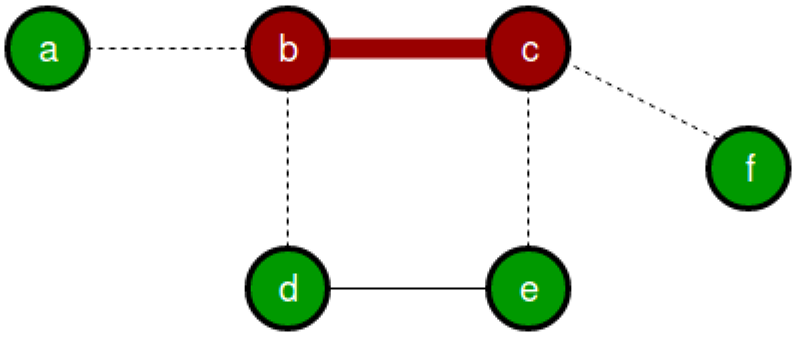
Approximate Algorithm for Vertex Cover:

```
1) Initialize the result as {}
2) Consider a set of all edges in given graph.  Let the set be E.
3) Do following while E is not empty
    a) Pick an arbitrary edge (u, v) from set E and add 'u' and 'v' to result
    b) Remove all edges from E which are either incident on u or v.
4) Return result
```

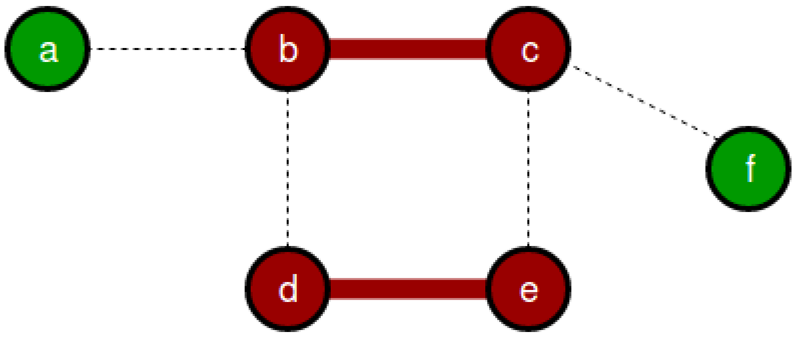
Below diagram to show execution of above approximate algorithm:



(a)



(b)



(c)

Minimum Vertex Cover is {b, c, d} or {b, c, e}

Applications of Randomized algorithms

- Randomized algorithms have huge applications in Cryptography.
- Load Balancing.

- Number-Theoretic Applications: Primality Testing
- Data Structures: Hashing, Sorting, Searching, Order Statistics and Computational Geometry.
- Algebraic identities: Polynomial and matrix identity verification. Interactive proof systems.
- Mathematical programming: Faster algorithms for linear programming, Rounding linear program solutions to integer program solutions
- Graph algorithms: Minimum spanning trees, shortest paths, minimum cuts.
- Parallel and distributed computing: Deadlock avoidance distributed consensus.

Follow the URLs

https://www.afs.enea.it/nicosia/infoteorica/NP_Complete.pdf

https://www.youtube.com/watch?v=e2cF8a5aAhE

https://www.youtube.com/watch?v=dQr4wZCiJJ4